# Featured Model Types: Towards Systematic Reuse in Modelling Language Engineering

Gilles Perrouin
PReCISE Research Center
University of Namur, Belgium
gilles.perrouin@unamur.be

Moussa Amrani
PReCISE Research Center
University of Namur, Belgium
moussa.amrani@unamur.be

Mathieu Acher
IRISA, University of Rennes I, France
mathieu.acher@irisa.fr

Benoit Combemale
INRIA Rennes, France
benoit.combemale@inria.fr

Axel Legay
INRIA Rennes, France
axel.legay@inria.fr

Pierre-Yves Schobbens
PReCISE Research Center, University of Namur, Belgium
pierre-yves.schobbens@unamur.be

## ABSTRACT

By analogy with software product reuse, the ability to reuse (meta)models and model transformations is key to achieve better quality and productivity. To this end, various opportunistic reuse techniques have been developed, such as higher-order transformations, metamodel adaptation, and model types. However, in contrast to software product development that has moved to systematic reuse by adopting (model-driven) software product lines, we are not quite there yet for modelling languages, missing economies of scope and automation opportunities. Our vision is to transpose the product line paradigm at the metamodel level, where reusable assets are formed by metamodel and transformation fragments and "products" are reusable language building blocks (model types). We introduce *featured model types* to concisely model variability amongst metamodelling elements, enabling configuration, automated analysis, and derivation of tailored model types. We provide a wish list of software engineering activities to work with featured model types.

## CCS Concepts

**•Software and its engineering → Model-driven software engineering; Domain specific languages; Software product lines;**

## Keywords

Model Typing, Metamodelling, Reuse

## 1. INTRODUCTION

Model-Driven Engineering (MDE), and in particular the area of Domain-Specific Modelling Languages (DSMLs), leads language designers to capture the language's concepts in metamodels. Models, representing particular instances of metamodels, can then be manipulated through transformations to perform translations to other languages, analyses (e.g., static semantics checking, or semantic correctness), evolution (refactorings, etc.) or queries (e.g., metrics or view extraction).

Yet as a DSML evolves, all the associated transformations need to evolve accordingly. Given the amount of effort that can be put into transformation design, reuse is indispensable. Research on transformation reuse engendered many interesting techniques, that can be roughly sorted in two categories. The first category focusses on adapting the transformations themselves: refactoring of model transformations (e.g. [36]), higher-order transformations ( [41]), model transformations "lifting" so that they can be applied on all the products of a product line [35] by changing their execution semantics. The second category reuses transformations without changing their internals, by acting on the model elements to which these model transformations apply: Sen *et al.* extract the concepts used by a model transformation and adapt the target metamodel to match those concepts [38], Cuadrado *et al.* defines generic model transformations [9], generalizing model refactorings proposed by Moha *et al.* [26]. Genericity is obtained via the definition of templates [10], concept metamodels [9] or model types [39], abstract structures that can be matched against actual target metamodels. There are also hybrid approaches that mixes the two categories, e.g., to deal with heterogeneous metamodels [11].

As surveyed by Kusel *et al.* [22], transformation reuse approaches can be further categorized along three dimensions: the transformation's genericity , its scope (intra/inter-metamodel), and its granularity (large/small parts of the transformation implementation). Picking up a particular combination on these dimensions provides different scenarios for reuse. The authors conclude that despite the large variety of mechanisms, transformations are reused opportunistically, and on a small scale. Three issues, identified by the authors, are independent of the transformation language used for expressing transformations:

**I1** – *Insufficient abstraction from metamodels*, meaning that MDE lacks standard hierarchies of metamodels and transformations;

**I2** – *Missing repositories for selection*, meaning that repositories of reusable artefacts, both at a fine-grained and coarse-grained level for transformation, are missing for



*MiSE'16, May 16-17, 2016, Austin, TX, USA*



DOI: http://dx.doi.org/10.1145/2896982.2896987

supporting large-scale transformation reuse;

**I3** – *Lack of meta-information for selection*, meaning that it is difficult to reuse transformations as is without knowing their internals, which can be overcome by providing appropriate meta-information.

In this paper, we form the vision that *adopting the software product line paradigm at the language engineering level* is a promising way to overcome these barriers. Software Product Lines (SPLs) [31] promote the systematic reuse of software assets belonging to a given domain. They do so by carefully managing the assets base, providing compact representations (feature diagrams [20]) of the commonalities and variabilities amongst products derivable from this asset base, and (semi)-automated combination of these assets in a product based on the selection its user relevant characteristics (features). SPL engineering distinguishes the construction and the management of the assets (*Domain Engineering*, or DE) and the building of products on top of the assets infrastructure (*Application Engineering*, AE). Transposing the SPL paradigm for MDE means that the assets base is formed by reusable metamodel and model transformation fragments and the "products" are DSMLs building blocks. As a long-term goal, engineers should be able to derive new DSMLs the same way we configure our cars and clothes on the Internet today.

To this end, we introduce the notion of *Featured Model Types* (FMTs). This structure can be seen as a kind of metamodel that integrates the variability of a whole family of metamodels in the same place, as well as a catalog of associated transformations applicable depending on certain combinations of features. Therefore, a FMT provides an abstract and compact way to describe metamodel hierarchies (addressing **I1**). Explicit variability modelling amongst metamodelling elements allows reuse both at the coarse-grained and fine grained level, thus providing incentives to manage repositories of specialised domain assets (tackling **I2**). Variability models can serve to engineer user-friendly configuration interfaces [6], allowing language engineers to express their needs in a simple but consistent way, through formal reasoning on the FMT (**I3**). Once choices enacted, tailored model types can be automatically derived [29, 30] prior to their matching and alignment on target metamodels (or use as a new DSML). Thus, our vision does not try to invent new reuse mechanisms but rather encompasses existing ones in a systematic product line development paradigm for metamodels, expecting the same successes this paradigm achieved at the model/code level.

We start by presenting the necessary background on Model Type and Feature Diagrams in Section 2. Then, Section 3 defines our notion of Featured Model Type. Section 4 provide our FMTs operations wish list. Section 5 discusses related approaches and finally, Section 6 wraps up the paper.

# 2. BACKGROUND

## 2.1 Model Type

A Domain-Specific Modelling Language (DSML) captures the knowledge of an expertise domain through high-level concepts closer to what experts manipulate in a daily basis, rather than constructions tailored to a particular technical solution. By trading generality for specificity, a DSML allows a higher level of automation: usually, transformations are associated with the DSML definition to analyse, verify and validate it, test instances, translate them into appropriate formats, or extract relevant information, such as metrics or views [23], thus simplifying the daily manipulation of instances. By nature, a DSML is built for a limited audience; however, many DSMLs are often specified independently of each other, although they share many basic concepts, but are syntactically incompatible. As a consequence, the associated transformations are also redefined for each particular purpose, thus hindering their reuse. As depicted in Figure 1, the issue is sometimes purely syntactic: metamodels can use different names for the same concepts, or topologically arrange them differently, or add details irrelevant for some manipulations (e.g., semantic details wrt. syntactic transformations). Generally, a model can conform to only one metamodel, the one used to build it.

The notion of Model Type circumvents these limitations: certain metamodels are considered as type, to which client metamodels have to be realigned in order to reuse the associated transformations.

DEFINITION 1 (MODEL TYPE). *A model type* $MT \in \mathcal{T}$ *is a triplet* $MT = (\mathsf{Name}, \mathsf{MM}, \mathsf{T})$ *where* $\mathsf{Name}$ *is the (model) type name,* $\mathsf{MM}$ *is a* MOF*-compliant [28] metamodel, and* $\mathsf{T}$ *is the set of associated transformations (defined as a* MOF*-like operation: name, return type and typed parameters).*

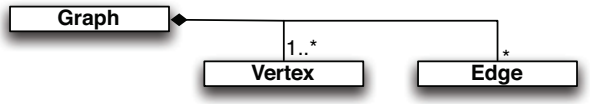
The transformations $\mathsf{t}_1, \ldots, \mathsf{t}_n \in \mathsf{T}$ are specified with the vocabulary defined in $\mathsf{MM}$: the concepts and the navigation necessary for specifying the building blocks of transformations (either rules in graph-based transformations; or statements and expressions in meta-programmed transformations). Following the metamodel-based approach, a client metamodel $\mathsf{MM}'$ that needs to reuse $\mathsf{t}_i$ has to be aligned (also called *adaptation* in [15]) to $\mathsf{MM}$, i.e. $\mathsf{MM}'$ has to be transformed in order to *match* $\mathsf{MM}$ in its vocabulary (i.e., class and class properties' names) and topology (i.e. how classes are connected), so that $\mathsf{t}_i$ can be used on models complying to $\mathsf{MM}'$. This compares to multi-sorted algebras: to reuse operations associated with a sort, a term needs to prove that it belongs to that sort [8].

The *transformations* associated with an MT usually rely on the the *operations* defined inside the classes of the metamodel. These operations can, in turn, be shared among some of the transformations, leading to so-called *intra-level* reuse [22]. However, this granularity is currently not well organised. In particular, operations have to precisely specify which structural elements they use from the MT, so that sharing them and reusing them among transformations can be performed in a safe way. Several techniques, mostly based on static discovery of metamodel footprints are already available for that purpose [18].

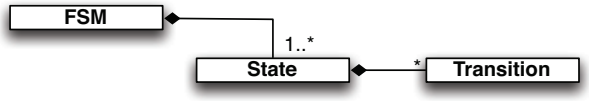
## 2.2 Feature Diagrams

*Feature Diagrams*, (FDs) introduced by Kang et al. [20], compactly and abstractly represent commonalities and variabilities of all the members of an SPL in terms of features: some of them are common to all products but some of them are only shared by few products. Each SPL member is thus uniquely identified by a combination of features. An FD organises features in a taxonomy, graphically depicted as a tree-like structure, in which the selection of leaf features implies the inclusion of their parents and sibling selection is controlled by operators. FDs can be formalised [37] to ease

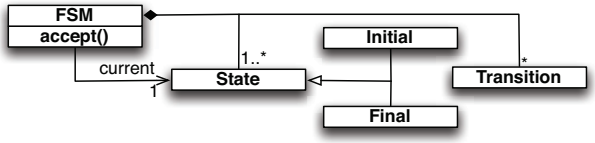

(a) Base metamodel FSM.




(b) Graph-Based representation of an Fsm.




(c) An Fsm with a different topology.




(d) Extension of the FSM metamodel with execution details.

Figure 1: Variations on the FSM Dsml: (a) Base Metamodel (for simplifying the metamodels, references between State and Transition are omitted here and in the remainder of the paper); (b) Variation on the concepts' names (Vertex and Edge instead of State and Transition); (c) Different topology (State, instead of FSM in (a), contains Transition); (d) Additional details for execution purpose.

reasoning about the whole Spl [5] and to support derivation techniques [29, 30].

Definition 2 (Feature Diagram [37]). *A feature diagram* $\mathsf{FD} \in \mathcal{F}$ *is a quadruplet* $\mathsf{FD} = (\mathsf{N}, \mathsf{P}, \lambda, \mathsf{DE}, \phi)$ *where* $\mathsf{N}$ *is a set of nodes and* $\mathsf{P} \subseteq \mathsf{N}$ *a set of primitive nodes, and with a distinguished* root *node* $r \in \mathsf{N}$, $\mathsf{DE} \subseteq \mathsf{N} \times \mathsf{N}$ *a set of (directed) decomposition edges (noted classically* $n_1 \longrightarrow n_2$ *iff* $(n_1, n_2) \in \mathsf{DE}$*), and* $\phi$ *a set of boolean formulæ over* $\mathsf{N}$ *expressing constraints. The function* $\lambda \colon \mathsf{N} \to \mathsf{NT}$ *labels each node with an n-ary boolean operator* $\mathsf{op}_n \in \mathsf{NT}$ *that indicates through which operator child nodes are related to their parent.* FD *is well-formed iff the graph represented by* FD *is acyclic and only possesses one* root *(i.e. the top-level entry node) and the arity of each operator* $\mathsf{op}_n$ *is respected.*

The node type set NT includes the typical operators encountered in classical feature diagrams notations (cf. [37]). A configuration $c$ is a set of (selected) primitive nodes (i.e. $c \in \wp(\mathsf{P})$, where $\wp$ is the powerset operator). FD thus induces sets of configurations (i.e. elements of $\wp(\wp(\mathsf{P}))$). The semantics of an Fd is the set of configurations in which each member satisfies the following conditions: the root is selected; the selected nodes should have their operators evaluate to true.

A graphical representation of an Fd can be found in Figure 2. From the root feature r, we notice its 3 optional (lollipop) children, h, x, t. The requires graphical constraints mandates that each time t is chosen x will be too.

## 3. FEATURED MODEL TYPES

This section proposes a new representation of Mt artifacts, called *Featured Model Type* (Fmt), for representing both the structural and the transformational artifacts variations. Fmts are intended as a compact representation that provides better performance for a large variety of analyses, that expresses choices for new, alternate Mts or transformation reuses. This section defines Fmts and illustrates how they can used to easily derive new Mt from a configuration.

An Fmt basically consists of an annotated metamodel: this metamodel contains all structural variants of the defining metamodels in the model type family, and each of its elements is annotated by a boolean feature expression describing which product(s) will map to this model element.

Definition 3 (Featured Model Type). *A Featured Model Type* $\mathsf{FMT} \in \mathcal{FT}$ *is a tuple* $\mathsf{FMT} = (\mathsf{Name}, \mathsf{MM}, \mathsf{FD}, \mathsf{T}, \mu)$ *where* $(\mathsf{Name}, \mathsf{MM}, \mathsf{T}) \in \mathcal{T}$ *is a model type,* $\mathsf{FD} \in \mathcal{F}$ *is a feature diagram and* $\mu \colon \mathsf{MElement} \cup \mathsf{T} \to \mathbb{B}(\mathsf{N}, \wedge, \vee, \neg)$ *a partial function mapping metamodel elements (classes, operations and class properties) and* Fmt *transformations to a boolean formulæ over the node set* N *of* FD.

Figure 2 depicts an Fmt for the Finite State Machines (Fsms) domain. The name FMT_FSM appears in the first compartment. The second compartment contains the metamodel elements present in each Fsm variant. The fourth one contains the associated transformations. All together, they form a valid Mt. A core metamodel, in black (with a root container FSM containing States and Transitions — Note we omitted associations between them to enhance readability), is progressively extended to represent three other variants. In orange, relevant elements are added in order to deal with executability: a new transformation accept becomes available, which is defined using new concepts: the current State maintained during computation, which starts in Initial states and ends in Final ones. In blue, the time dimension becomes available within States, exhibiting two associated transformations: wcet for computing worst-case execution times; and accept. Note that accept is syntactically different from the previous one, thus using different transformation units. Finally in green, nested States become available for representing hierarchical Fsms that can eventually be flattened.

The third compartment describes an Fd for the structural elements, and each transformation is associated with the relevant features (r is the root; x, t and h represent respectively executability, time and hierarchical variants of the Fsm). The mapping function $\mu$ annotates each metamodel element in $\mathsf{MM}_{\mathsf{FMT\_FSM}}$ (including Fmt transformations) with a boolean expression indicating whether those features need these metamodel elements or not, i.e. metamodel elements should be present for a given feature (colors were used in Figure 2 instead of annotations to ease reading).

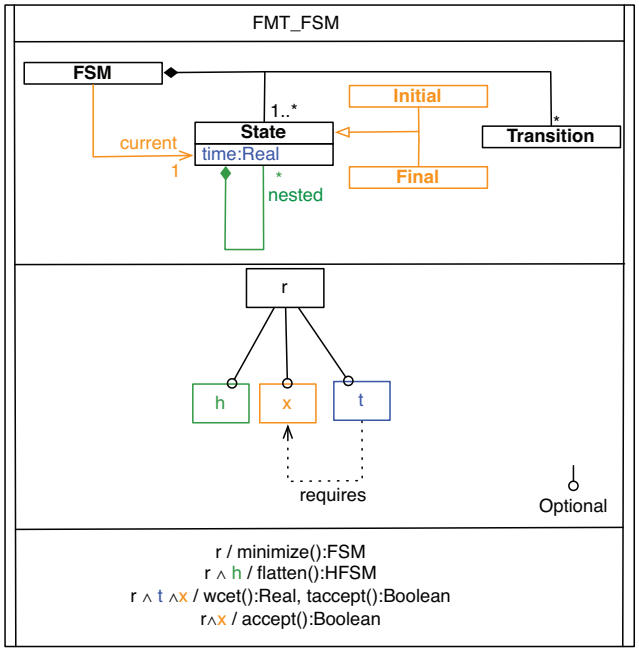



Figure 2: An FMT for the Finite State Machine domain: executability, hierarchy and time concerns.

Reuse at the intra-level granularity [22], i.e. at a finer grain than transformations (rules sets in graph transformations; operations like fireable in metaprogrammed transformations, used to specify accept), is achieved with the very same mechanism: features decorate those fine-grained artifacts, describing their inter-relationships and the way they integrate within transformations through constraints. Typically, a transformation will *require* the operations / rules it uses, which in turns will reference the structural elements in the metamodel they need. For building an FMT at this fine-grained level, we can rely on static analysis techniques for computing call graphs and transformation footprints [18].

Once an FMT is built for a specific domain, like the FSM in our example, it becomes possible to derive a minimal, yet complete metamodel suitable for reusing designated transformations. Figure 3 shows such a metamodel, for a configuration composed of flatten and accept. Since the derived metamodel also includes the core ingredients of an FSM, the derivation mechanism also includes minimize as a candidate transformation for reuse. Fine-grained operations could also be returned, assuming the FMT includes them.

# 4. WORKING WITH FMT — A WISH-LIST

This section presents a wish list of the main operations needed in FMTs environments to enable systematic reuse. For each operation, we provide a description as well as the advances and challenges involved in its realization. The first two operations focus on domain engineering or how to create an operational FMT infrastructure, while the last four target application engineering or how to exploit FMT to derive tailored model types.

## 4.1 Building FMTs

**Purpose** A central task is to elaborate FMTs. A manual building is a first possible solution. As we illustrated in

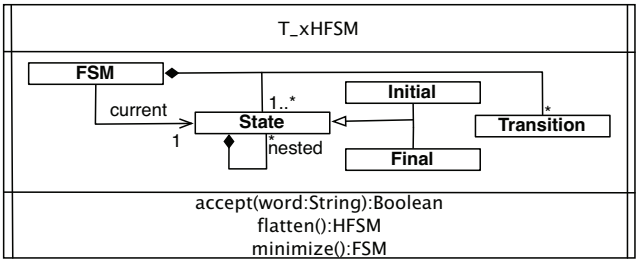



Figure 3: Derived FMT based on selecting transformations flatten() and accept().

Section 3, we can start from an MT (that will form the root of the FMT) and incrementally add metamodel elements and transformations to support variants. This scenario is relevant when we want to build a family of model types from the onset. Yet, as for SPLs, this is rarely the case. Indeed, reuse opportunities are often discovered when there is a set of related existing products we want to reuse from. In that case, we need to transition from a set of "clone" metamodels to an FMT. A clone management framework offers primitives like findFeatures, same?, or merge [34]. These primitives can assist domain engineers in the synthesis of FMTs. Model-driven Product line synthesis [24] is also to be considered here.

**Challenges** Manual construction of FMTs is error-prone: it is easy to forget a constraint at the FMT level, or boolean formulæ can be wrongly specified. The possible consequence is to obtain an unsound or incomplete FMT, hindering a proper derivation or analysis. We should therefore provide FMT synthesis primitives (computing features and their dependencies, merging similar metamodel elements, etc.). These primitives have to guarantee correct by construction FMTs, relying on validation operations (see below). Another challenge is the evolution of FMTs: how do we integrate new elements in an existing FMT? Several strategies can be assessed such as offering these elements, tagged by one unique feature, in a mutually exclusive form first. Progressively, these elements will be annotated with other features enabling finer-grained reuse. Our FMT proposal currently considers transformations as atomic, a final challenge is to extend FMTs to transformation internals, in a similar way to what is proposed by Strüber *et al.* [40].

## 4.2 Validating FMTs

**Purpose** At any step of the FMT lifecycle, domain engineering validation activities are required. A common check we want to perform is to ensure that there is no combination of model transformations that can lead to an inconsistent MT. Some inconsistencies, as mentioned earlier, are structural: conflicting names, references with incompatible multiplicities, or attributes with different types. Some others are related to transformation semantics: what does it mean to execute accept on a T_xHFSM, since accept is not intended to handle nested states? As for SPL, the goal is to take advantage of commonalities and variabilities to save on the analysis effort by avoiding to enumerate any possible MT, since the number of possibilities grows exponentially with the number of features. Kästner *et al.* proposed in [21] a technique called variability-aware typechecking, to deal with structural inconsistencies. Regarding semantics incon-

sistencies, depending on the nature of the transformation language, either variability-aware verification [7] or combinatorial interaction testing approaches [19] may be adapted to that task.

**Challenges** All family-based analyses can be affected by scalability issues. We believe that FMT typechecking is feasible on such a large scale, since Kästner *et al.* [21] validated their technique on large code bases. For semantic issues, all will depend on the granularity of the verification. For complex transformations, developing a suite of reusable tests, to be run during application engineering, may be complementary.

## 4.3 Configure a new MT product

**Purpose** This operation aims at assisting the user for selecting the appropriate fragments suitable for his needs. As available from the FMTs, the user can either select transformations he knows he will have to reuse, or structural fragments from MT that capture the concepts of his language.

**Challenges** Similarly to PL configuration, several design choices appear when building configurators based on FMTs. When a configuration is complete, i.e. all features have been resolved, it only remains to check for the configuration's validity. However, a configuration may be only partial, because the user is not sure which choices to select for certain features, or because some choices are left open for exploration purposes. Presenting the remaining unresolved features in a useful way for helping the user to actually perform the desired transformations can be challenging. On another hand, a configurator can be designed in a guided way, pruning choices that become impossible when new features are selected, or designed in a more permissive way (e.g., for expert users), but bundled with stronger analysis capabilities for checking configurations at the end of the process. It is an open issue to know which scheme works best for FMTs, and there are chances that it is application-specific (or depends on the engineer expertise). In all cases, adequately documenting transformations and MTs is a key enabler for performing meaningful choices, as already noticed by Kusel *et al.* [22].

## 4.4 Derive an MT product from a configuration

**Purpose** From a validated configuration, it is important to be able to derive a product. In our case, the product corresponds to an MT, to which the user has to align his metamodel in order to reuse the associated transformations.

**Challenges** When several FMTs already exist for the domains targeted by the user, deriving a product consists in simply reusing the existing technology for software product lines, except that products are in this context FMTs, i.e. metamodels accompanied with transformations.

For example, Figure 3 shows a derived FMT, obtained from selecting the flatten() and accept() transformations: the derived MT shows metamodel elements that do not appear within the original MTs. The user has then to align his client metamodel with the obtained MT in order to be able to fully reuse the selected transformations. Several techniques already exist for that purpose, e.g. based on standard languages like OCL [12].

## 4.5 Validate an MT Product

**Purpose** Once the MT has been derived, there may remain individual validation operations to perform, which are too complex to be assessed for all possible MT at the domain engineering level. This can be the case for ensuring that every transformation of the newly build MT works as intended by their specifications. Depending of the model transformation nature (model-to-model, model-to-text) verification [3] and testing techniques [14] may also be applied for an individual MT product.

**Challenges** Naturally, we do not expect verification properties and tests to be rewritten for each derived MT, but to reuse them. This raises the challenge of defining such validation artifacts on FMT, i.e. envisioning all the semantics variations a transformation may support when used with different combination of features. If these artifacts existed prior to the inclusion of the transformation in an FMT, then lifting might be an option [35]. Another question that arises is the validation of generic FMT test suites. Techniques such as mutation analysis [27] may also need to be lifted to the FMT formalism. As mentioned by Kusel *et al.* [22] we may still need to write "integration tests" (i.e., detecting unexpected interactions amongst model transformation) for a derived MT.

## 4.6 MT Matching and Customization

**Purpose** Once derived and validated, the MT product is ready to be used on concrete metamodels. If the MT is derived from a complete configuration, all choices have been resolved and existing matching techniques apply [15,39]. Depending on the context, transformations' semantics, more or less strict matching relationships between the derived MT product and target metamodel may be enforced.

**Challenges** Challenges arise when: *(i)* the configuration is partial (in that case, derived product still contains some unresolved features, technically being an FMT) or *(ii)* further customization is required on the model type, for example to support unexpected model elements or/and transformation. Answering to *(i)* involves revisiting exiting matching relationships to cover the presence of variability. The second challenge requires the customization is done in a disciplined way [29], by defining additional constraints on the derived MT [46], possibly inherited from FMT, to ensure that the customization does not break the derive MT properties and transformation semantics.

# 5. RELATED WORK

Model Types were initially introduced by Steel [39], and the notion was further explored by Guy *et al.* [15] and Degueule *et al.* [13]. It fully exploits the characteristics of MOF, and has therefore no equivalent in Graph-Based Model Transformations. Zschaler [46] proposed an unified representation of model types based on constraints. This paper proposes to enrich MTs with variability (features) in order to manage an MT hierarchy. Variability for metamodels has already been explored in [30]. Techniques were presented to weave variability in metamodel constructs. The goal was to add more flexibility in the usage of a given metamodel. In our approach we aim to manage a set of existing metamodels. Our vision also encompasses the consistent management of transformations associated with MTs.

Several contributions applied variability management to programming or modelling languages: Vacchi *et al.* [42] for reverse engineering techniques with Neverlang, a modular language implementation framework; Cengarle *et al.* for describing variations of a base language in MontiCore; or Haugen *et al.* [16] for modelling possible variations in DSMLs in CVL (Common Variability Language); or White *et al.* [43] for improving reusability of features among a language family, among others. Our vision proposes an explicit, concise formalism for FMTs for managing a family of languages (or metamodels); a potential target language can be Clafer [4, 16], a metamodelling language mixing features and classes.

Inferring product lines models has already been studied [2, 17, 24, 25, 32, 33, 42, 44, 45]. Holthusen et al. [17] propose to mine a so-called family model for function block diagrams. Martinez et al. propose feature mining and visualization techniques for managing variability of model variants [25]. Rubin and Chechik propose to combine related product (models) into product lines (models annotated with presence conditions) [32, 33]. For example, it is possible to infer from UML models a product line representation. In their case, all input models are instances of the *same* metamodel. Hence their approach does not consider the inference (or synthesis) of families of languages. Zhang *et al.* describe comparison techniques to synthesize model-based product lines [44] as CVL models [16]. An extension to augment product lines and thus to support the incremental synthesis of product lines is subsequently proposed in [45]. The specific problem of synthesising FMT from a set of input metamodels has not been explored so far and remains a challenge (see Section 4.1).

Salay et al. propose to make applicable transformations defined on individual products on the whole product line, by changing the execution semantics of the transformation (defined in terms of rewriting rules) [35]. In our work, we aim to have a catalog of transformations that can be applied to any "product" metamodel.

# 6. CONCLUSION

In this paper, we described a vision in which model types should be engineered as families rather than independently. We introduced the formal notion of featured model type to this end and exhibited how techniques originally developed in the SPL community can be applied to FMT to provide automated support for model type derivation depending on architectural / behavioural concerns as well as family-wise analyses. We first plan to realise our vision using DSML frameworks such as Clafer [4], MELANGE [13], and/or FAMILIAR [1] to specify FMT. The next item on the research agenda concerns the semi-automated synthesis of FMT from a collection of existing artifacts. Finally we will also investigate type-checking and consistency analyses at the FMT level to guarantee certain properties on derived model types.

# 7. REFERENCES


[1] M. Acher, P. Collet, P. Lahire, and R. B. France. Familiar: A domain-specific language for large scale management of feature models. *Science of Computer Programming*, 78(6):657–681, 2013.

[2] M. Acher, B. Combemale, P. Collet, O. Barais, P. Lahire, and R. B. France. Composing your compositions of variability models. In *MODELS*, pages 352–369, 2013.

[3] M. Amrani, B. Combemale, L. Lucio, G. M. K. Selim, J. Dingel, Y. L. Traon, H. Vangheluwe, and J. R. Cordy. Formal verification techniques for model transformations: A tridimensional classification. *Journal of Object Technology*, 14(3):1:1–43, 2015.

[4] K. Bak, K. Czarnecki, and A. Wasowski. Feature and meta-models in clafer: mixed, specialized, and coupled. In *SLE*, pages 102–122, 2011.

[5] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

[6] Q. Boucher, G. Perrouin, and P. Heymans. Deriving configuration interfaces from feature models: a vision paper. In *VaMoS*, pages 37–44. ACM, 2012.

[7] A. Classen, M. Cordy, P. Schobbens, P. Heymans, A. Legay, and J. Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE TSE*, 39(8):1069–1089, 2013.

[8] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti Oliet, J. Meseguer, and C. Talcott. *All About* MAUDE. Springer, 2007.

[9] J. S. Cuadrado, E. Guerra, and J. De Lara. Generic model transformations: write once, reuse everywhere. In *Theory and practice of model transformations*, pages 62–77. Springer, 2011.

[10] J. de Lara and E. Guerra. Reusable graph transformation templates. In *Applications of Graph Transformations with Industrial Relevance*, volume 7233 of *LNCS*, pages 35–50. Springer, 2011.

[11] J. de Lara and E. Guerra. Towards the flexible reuse of model transformations: A formal approach based on graph transformation. *Journal of Logical and Algebraic Methods in Programming*, 83(5–6):427 – 458, 2014.

[12] J. de Lara, E. Guerra, and J. S. Cuadrado. A-posteriori typing for model-driven engineering. In *MODELS*, pages 156–165. IEEE, 2015.

[13] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J.-M. Jézéquel. Melange: A meta-language for modular and reusable development of dsls. In *SLE*, pages 25–36. ACM, 2015.

[14] F. Fleurey, J. Steel, and B. Baudry. Validation in model-driven engineering: testing model transformations. In *MODEVA*, pages 29–40, Nov 2004.

[15] C. Guy, B. Combemale, S. Derrien, J. R. Steel, and J.-M. Jézéquel. On model subtyping. In *ECMFA*, pages 400–415. Springer, 2012.

[16] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen. Adding standardized variability to domain specific languages. In *SPLC*, pages 139–148, 2008.

[17] S. Holthusen, D. Wille, C. Legat, S. Beddig, I. Schaefer, and B. Vogel-Heuser. Family model mining for function block diagrams in automation software. In *SPLC*, pages 36–43. ACM, 2014.

[18] C. Jeanneret, M. Glinz, and B. Baudry. Estimating footprints of model operations. In *International Conference on Software Engineering*, pages 601–610, 2011.

[19] M. F. Johansen, Ø. Haugen, and F. Fleurey. Properties of realistic feature models make

combinatorial testing of product lines feasible. In *MODELS*, pages 638–652. ACM/IEEE, 2011.

[20] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Nov. 1990.

[21] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM TOSEM*, 21(3):14, 2012.

[22] A. Kusel, J. Schönböck, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger. Reuse in model-to-model transformation languages: Are we there yet? SoSyM, 14(2):537–572, May 2015.

[23] L. Lúcio, M. Amrani, J. Dingel, L. Lambers, R. Salay, G. Selim, E. Syriani, and M. Wimmer. Model Transformation Intents and Their Properties. SoSyM, 2014.

[24] J. Martinez, T. Ziadi, T. Bissyandé, J. Klein, and Y. Le Traon. Automating the extraction of model-based software product lines from model variants. In *ASE*, pages 396–406. ACM/IEEE, Nov. 2015.

[25] J. Martinez, T. Ziadi, J. Klein, and Y. Le Traon. Identifying and visualising commonality and variability in model variants. In *ECMFA*, pages 117–131, 2014.

[26] N. Moha, V. Mahé, O. Barais, and J.-M. Jézéquel. Generic model refactorings. In *MODELS*, pages 628–643. ACM/IEEE, 2009.

[27] J.-M. Mottu, B. Baudry, and Y. Le Traon. Mutation analysis testing for model transformations. In *ECMDA-FA*, volume 4066 of *LNCS*, pages 376–390. Springer, 2006.

[28] OMG. Meta object facility (mof) core specification. Technical Report 06-01-01, OMG, January 2006.

[29] G. Perrouin, J. Klein, N. Guelfi, and J.-M. Jézéquel. Reconciling automation and flexibility in product derivation. In *SPLC*, pages 339–348, 2008.

[30] G. Perrouin, G. Vanwormhoudt, B. Morin, P. Lahire, O. Barais, and J. Jézéquel. Weaving variability into domain metamodels. SoSyM, 11(3):361–383, 2012.

[31] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[32] J. Rubin and M. Chechik. Combining related products into product lines. In *FASE*, pages 285–300, 2012.

[33] J. Rubin and M. Chechik. Quality of Merge-Refactorings for Product Lines. In *FASE*, pages 83–98, 2013.

[34] J. Rubin, K. Czarnecki, and M. Chechik. Managing cloned variants: A framework and experience. In *SPLC*, pages 101–110, 2013.

[35] R. Salay, M. Famelis, J. Rubin, A. Di Sandro, and M. Chechik. Lifting model transformations to product lines. In *ICSE*, pages 117–128. ACM, 2014.

[36] J. Sánchez Cuadrado and J. García Molina. Approaches for model transformation reuse: Factorization and composition. In *Theory and Practice of Model Transformations*, volume 5063 of *LNCS*, pages 168–182. Springer, 2008.

[37] P. Schobbens, P. Heymans, J. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007.

[38] S. Sen, N. Moha, V. Mahé, O. Barais, B. Baudry, and J.-M. Jézéquel. Reusable model transformations. SoSyM, 11(1):111–125, 2012.

[39] J. Steel and J.-M. Jézéquel. On model typing. *Software and Systems Modeling*, 6(4):401–413, 2007.

[40] D. Strüber, J. Rubin, M. Chechik, and G. Taentzer. Variability-based approach to reusable and efficient model transformations. In *FASE*, volume 9033 of *LNCS*, pages 283–298. Springer, 2015.

[41] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the use of higher-order model transformations. In *ECMDA-FA*, volume 5562 of *LNCS*, pages 18–33. Springer, 2009.

[42] E. Vacchi, W. Cazzola, B. Combemale, and M. Acher. Automating variability model inference for component-based language implementations. In *SPLC*, pages 167–176. ACM, 2014.

[43] J. White, J. H. Hill, J. Gray, S. Tambe, A. Gokhale, and D. C. Schmidt. Improving domain-specific language reuse with software product-line configuration techniques. *IEEE Software*, 26(4):47–53, July-Aug. 2009.

[44] X. Zhang, Ø. Haugen, and B. Møller-Pedersen. Model comparison to synthesize a model-driven software product line. In *SPLC*, pages 90–99. IEEE, 2011.

[45] X. Zhang, Ø. Haugen, and B. Møller-Pedersen. Augmenting product lines. In *Asia-Pacific Software Engineering Conference*, pages 766–771. IEEE, 2012.

[46] S. Zschaler. Towards constraint-based model types - a generalised formal foundation for model genericity. In *View-Based, Aspect-Oriented and Orthographic Software Modelling*, pages 11:11–11:18. ACM, 2014.